Enrolment No./Seat No_____

GUJARAT TECHNOLOGICAL UNIVERSITY

BE - SEMESTER–III (NEW) EXAMINATION – SUMMER 2024

Subject Code:3130703 Date:29-06-2024

Subject Name: Database Management Systems

Time:10:30 AM TO 01:00 PM Total Marks:70

Instructions:

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**
4. **Simple and non-programmable scientific calculators are allowed.**

			MARKS
Q.1	**(a)**	Define Logical Data Independence and Physical Data Independence.	**03**
	(b)	Draw and explain 3 layers of abstraction of DBMS	**04**
	(c)	Write relational algebra expression for following **<u>(query 1,2 and 3 carry one mark/query 4 & 5 carry two marks)</u>** Employee(eid, ename, salary, deptid) Department(deptid, dname) 1) List name of all employee 2) List all the departments with id > 5 3) List sum of the salary all employees. 4) Department wise count the number of employees 5) Find the name of employee earning highest salary.	**07**
Q.2	**(a)**	List various symbol used for ER diagram	**03**
	(b)	Define referential integrity constraint with example.	**04**
	(c)	List the Armstrong axioms used to compute closure of the set of functional dependency.	**07**
		OR	
	(c)	What is decomposition? Explain dependency preserving decomposition with the help of example.	**07**
Q.3	**(a)**	Give syntax of command used to add a new column in a table.	**03**
	(b)	Define indexing. How will it be useful for searching data?	**04**
	(c)	How can we measure cost of Join query? Explain using example.	**07**
		OR	
Q.3	**(a)**	Write an SQL command to modify the content of table.	**03**
	(b)	Explain sparse indexing with the help of diagram.	**04**
	(c)	Draw and explain the query processing steps	**07**
Q.4	**(a)**	What is relational model? Explain schema diagram using example.	**03**
	(b)	Define transaction and its properties?	**04**
	(c)	Write a note on view serializability	**07**
		OR	
Q.4	**(a)**	With the help of example explain Generalization..	**03**
	(b)	Define various approach used for Log based recovery	**04**
	(c)	List and explain various types of Locks used for concurrency control.	**07**

Q.5 **(a)** How can we print output in PL/SQL block? Give example. **03**

(b) Explain cursor and its types. **04**

(c) Write SQL queries for the following **07**

(query 1,2 and 3 carry one mark/query 4 & 5 carry two marks)

Employee(eid, ename, salary, deptid)

Department(deptid, dname)

1. List name of all employee
2. List all the departments with id > 5
3. List sum of the salary all employees.
4. Department wise count the number of employees
5. Find the name of employee earning highest salary.

OR

Q.5 **(a)** Define encryption and decryption. **03**

(b) List various types of Trigers. **04**

(c) What is role? How can we authorize a user using grant and revoke command? Give example. **07**

****

Q.1 (a) Define Logical Data Independence and Physical Data Independence. (3 mark)

Introduction / Definition:
In database management systems (DBMS), **data independence** refers to the ability to **change the schema at one level without affecting the schema at the next higher level**. It is a key feature of DBMS that allows flexibility and reduces maintenance efforts. There are two types of data independence: **Logical Data Independence** and **Physical Data Independence**.

---

1. Logical Data Independence:

- Refers to the **ability to change the logical schema** (table structure, attributes, relationships) **without affecting the application programs** or external schema.
- Example: Adding a new column to a table or creating a new table does not require changes in the application using existing data.
- **Purpose:** Protects applications from frequent changes in **logical design**.

---

2. Physical Data Independence:

- Refers to the **ability to change the physical storage structure** (file organization, indexing, storage devices) **without affecting the logical schema or application programs**.
- Example: Changing from sequential file storage to indexed storage does not impact how the data is viewed or queried.
- **Purpose:** Ensures flexibility in **storage optimization** and improves performance without affecting users.

Key Points:

- Logical independence deals with **changes in table structures and relationships**.
- Physical independence deals with **changes in storage and access methods**.
- Both types are crucial for **maintaining database consistency, flexibility, and ease of maintenance**.

(b) Draw and explain 3 layers of abstraction of DBMS (4 mark)

Introduction / Definition:
Database Management Systems (DBMS) use **abstraction** to simplify interaction with data. Abstraction hides the complexities of data storage and provides **different views for different users**. There are **three levels of abstraction**: Physical Level, Logical Level, and View Level. These layers ensure **data independence, security, and ease of access**.

---

1. Physical Level (Lowest Level)

- Describes **how data is actually stored** in the database (files, indexes, storage paths).
- Focuses on **physical storage structures and access methods**.
- Provides details like **record placement, data blocks, and pointers**.
- Users **do not see this layer**; it is managed by the DBMS.

---

2. Logical Level (Conceptual Level / Middle Level)

- Describes **what data is stored and relationships among data**.

- Hides **physical storage details** from users.
- Defines tables, attributes, data types, and constraints.
- Provides a **community-wide view** for database designers and administrators.
- Example: Customer table has attributes Cust_Id, Name, Age, Address, Salary.

---

3. View Level (Highest Level / External Level)

- Describes **how data is viewed by individual users or applications**.
- Each user can have **customized views** based on their needs.
- Hides unnecessary details and **presents only relevant data**.
- Example: HR department may see employee name and salary, but not personal address.

Key Points / Advantages:

- **Data Abstraction** simplifies database usage.
- Provides **data security** by restricting access at the view level.
- Supports **logical and physical data independence**.
- Allows multiple users to interact with the database **without interfering with each other**.

Diagram (Described in Words):

- **Top Layer (View Level):** User-specific views → Shows only required data.

- **Middle Layer (Logical Level):** Conceptual schema → Defines tables, attributes, and relationships.
- **Bottom Layer (Physical Level):** Physical storage → Files, indexes, and storage details.
  (Imagine a three-layered pyramid with View at the top, Logical in the middle, and Physical at the bottom.)

(c) Write relational algebra expression for following (query 1,2 and 3 carry one mark/query 4 & 5 carry two marks) Employee(eid, ename, salary, deptid) Department(deptid, dname)

1) List name of all employee

2) List all the departments with id > 5

3) List sum of the salary all employees.

4) Department wise count the number of employees

5) Find the name of employee earning highest salary.

Relational Algebra Expressions

Given Tables:

- **Employee(eid, ename, salary, deptid)**
- **Department(deptid, dname)**

---

1) List name of all employees (1 mark)

$$\pi_{ename}(\text{Employee})$$

- **Explanation:** The **π (projection)** operator selects the ename column from the Employee table.

2) List all departments with deptid > 5 (1 mark)

$$\sigma_{deptid>5}(\text{Department})$$

- **Explanation:** The **σ (selection)** operator retrieves rows where deptid > 5.

---

3) List sum of the salary of all employees (1 mark)

$$\gamma_{\text{SUM(salary)}}(\text{Employee})$$

- **Explanation:** The **γ (aggregate)** operator calculates the sum of the salary attribute for all employees.

---

4) Department-wise count of number of employees (2 marks)

$$\gamma\text{deptid,COUNT(eid)}(\text{Employee})$$

- **Explanation:**
  - γ is the **grouping operator**.
  - Groups employees by deptid and counts eid in each department.
  - Returns **deptid** and number of employees in each department.

---

5) Find the name of employee earning highest salary (2 marks)

$$\pi_{ename}(\sigma_{salary=\text{MAX(salary)}}(\text{Employee}))$$

- **Explanation:**
  - σ selects rows where salary equals the maximum salary.

- π projects only the ename column.
- This retrieves the name(s) of employee(s) with the highest salary.

Q.2 (a) List various symbol used for ER diagram (3 mark)

Introduction / Definition:
An **ER diagram** is a graphical representation of entities, their attributes, and relationships in a database. It is widely used during **database design** to visually depict how data is organized and interrelated. ER diagrams use **specific symbols** to represent entities, relationships, and attributes.

Common Symbols in ER Diagram:

1. **Rectangle (Entity)**
   - Represents an **entity**, which is a real-world object or concept.
   - Example: Employee, Department.
2. **Oval / Ellipse (Attribute)**
   - Represents an **attribute** of an entity or relationship.
   - Example: Emp_Id, Name, Salary.
3. **Diamond (Relationship)**
   - Represents a **relationship** between two or more entities.
   - Example: Works_In between Employee and Department.
4. **Double Oval (Multivalued Attribute)**
   - Represents an attribute that can have **multiple values**.
   - Example: PhoneNumbers for an employee.
5. **Double Rectangle (Weak Entity)**

- Represents a **weak entity**, which depends on a strong entity for its identification.
- Example: Dependent of an Employee.

6. **Dashed Oval (Derived Attribute)**
   - Represents an attribute that can be **derived** from other attributes.
   - Example: Age can be derived from DateOfBirth.
7. **Lines / Connectors**
   - Connect entities to attributes or relationships.
   - Shows participation in relationships.
8. **Underlined Attribute**
   - Denotes the **primary key** of an entity.
   - Example: Emp_Id underlined in Employee.

Key Points:

- These symbols provide a **standardized way** to represent data structures visually.
- Help in understanding **data relationships and constraints** before actual database implementation.
- Make **database design clear, easy to communicate, and error-free**.

(b) Define referential integrity constraint with example. (4 mark)

Introduction / Definition:
In a relational database, **referential integrity** is a type of **integrity constraint** that ensures the **consistency and correctness of data across related tables**. It enforces that a **foreign key in one table**

must either **match a primary key in another table** or be **NULL**.
This prevents invalid data and maintains relationships between tables.

Explanation:

- A **foreign key** in a child table refers to the **primary key** in a parent table.
- Referential integrity ensures that **every value of the foreign key exists in the parent table**.
- Violating this rule can lead to **orphan records**, which are child records without a corresponding parent record.
- SQL provides options such as **ON DELETE CASCADE** or **ON UPDATE CASCADE** to handle changes in the parent table.

Example:

Tables:

- **Department(deptid, dname)** → deptid is primary key
- **Employee(eid, ename, deptid)** → deptid is foreign key referencing Department

SQL Constraint:

```
CREATE TABLE Department (
   deptid NUMBER PRIMARY KEY,
   dname VARCHAR2(50)
);


CREATE TABLE Employee (
```

```
    eid NUMBER PRIMARY KEY,
    ename VARCHAR2(50),
    deptid NUMBER,
    CONSTRAINT fk_dept FOREIGN KEY (deptid)
    REFERENCES Department(deptid)
);
```

- Here, every deptid in Employee **must exist** in the Department table.
- If someone tries to insert an employee with deptid = 10 but no such department exists, the DBMS will **reject the insert**.

Key Points / Advantages:

- Ensures **data consistency** across tables.
- Prevents **orphan records** in child tables.
- Maintains **correct relationships** in relational databases.

(c) List the Armstrong axioms used to compute closure of the set of functional dependency. (7 mark)

Introduction / Definition:
In a relational database, **functional dependency (FD)** describes a relationship between attributes where the value of one set of attributes determines the value of another. To **compute the closure of a set of functional dependencies** or to infer all possible FDs from a given set, **Armstrong's axioms** are used. These axioms are **sound and complete rules** for reasoning about functional dependencies.

Armstrong's Axioms

There are **three primary axioms** and **three additional rules derived from them**:

---

1. Reflexivity Rule:

- If Y is a subset of X, then $X \rightarrow Y$
- **Explanation:** Every set of attributes functionally determines its subset.
- **Example:** If $X = \{A, B\}$, then $\{A, B\} \rightarrow \{A\}$.

---

2. Augmentation Rule:

- If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
- **Explanation:** Adding the same attributes to both sides of a dependency preserves the dependency.
- **Example:** If $A \rightarrow B$, then $AC \rightarrow BC$.

---

3. Transitivity Rule:

- If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- **Explanation:** Functional dependency is transitive.
- **Example:** If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

Additional Rules (Derived from the above)

4. Union Rule:

- If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
- **Explanation:** Two dependencies with the same determinant can be combined.

5. Decomposition Rule:

- If X → YZ, then X → Y and X → Z
- **Explanation:** A dependency on multiple attributes can be broken down into multiple dependencies.

6. Pseudotransitivity Rule:

- If X → Y and WY → Z, then XW → Z
- **Explanation:** Combines transitivity and augmentation in a single step.

Key Points / Features:

- **Soundness:** All derived FDs are valid.
- **Completeness:** All valid FDs can be derived using these axioms.
- Used for **computing closure**, **normalization**, and **schema design**.

Example of Computing Closure:

- Given FDs: {A → B, B → C}
- Using **Transitivity**, A → C
- Using **Union**, A → BC

OR

(c) What is decomposition? Explain dependency preserving decomposition with the help of example. (7 mark)

Introduction / Definition:
In database design, **decomposition** refers to the process of **breaking a relation into two or more smaller relations** while preserving

important properties such as **lossless join** and **dependency preservation**. Decomposition is commonly used during **normalization** to eliminate redundancy, avoid anomalies, and maintain **data integrity**.

Dependency-Preserving Decomposition

Definition:
A decomposition is said to be **dependency-preserving** if all **functional dependencies** (FDs) from the original relation can be enforced by applying FDs **only on the smaller relations** after decomposition, without needing to join them back together.

Importance:

- Ensures **constraints and rules** on data are still enforceable after decomposition.
- Reduces the need for expensive **joins** to check functional dependencies.
- Helps maintain **data consistency and integrity**.

Steps / Procedure:

1. Identify all **functional dependencies (FDs)** in the original relation.
2. Check if the relation violates **normal forms** (1NF, 2NF, 3NF, BCNF).
3. Decompose the relation into smaller relations to **remove redundancy** and satisfy the desired normal form.
4. Ensure the decomposition is **lossless** (can reconstruct original relation using joins).

5. Check if the decomposition is **dependency-preserving**. If yes, all original FDs can be enforced on the new relations.

Example:

Original Relation:
R(A, B, C) with FDs:

- A → B
- B → C

Step 1: Identify FDs violating BCNF

- B → C violates BCNF because B is not a superkey.

Step 2: Decompose Relation

- Decompose R into:
    1. R1(B, C)
    2. R2(A, B)

Step 3: Check Dependency Preservation

- Original FDs: A → B and B → C
- A → B is preserved in R2
- B → C is preserved in R1
- ✅ All FDs are preserved without joining R1 and R2

Result:

- Decomposition is **lossless** and **dependency-preserving**.

Key Points / Advantages:

- Reduces **redundancy and anomalies** (insertion, update, deletion).
- Ensures **data integrity** through preserved dependencies.
- Makes database **efficient and consistent**.

Disadvantages:

- Sometimes achieving both **lossless and dependency-preserving** decomposition may not be possible for all relations.
- May require **more joins** in queries if relations are decomposed excessively.

Q.3 (a) Give syntax of command used to add a new column in a table. (3 mark)

Adding a New Column in a Table

Introduction / Definition:
In SQL, the structure of a table can be **modified after creation** using the ALTER TABLE command. One common operation is to **add a new column** to an existing table. This allows the database to **accommodate new data requirements** without dropping or recreating the table.

Syntax:

ALTER TABLE table_name

ADD column_name datatype [constraints];

Explanation of Syntax:

- ALTER TABLE table_name → Specifies the table you want to modify.

- ADD → Keyword used to add a new column.
- column_name → Name of the new column to be added.
- datatype → Specifies the type of data the column will store (e.g., NUMBER, VARCHAR2, DATE).
- [constraints] → Optional; can include NOT NULL, UNIQUE, DEFAULT, etc.

Example:

Suppose we have a table Employee(eid, ename, salary) and we want to add a deptid column:

ALTER TABLE Employee

ADD deptid NUMBER;

Explanation:

- Adds a new column deptid of type NUMBER to the Employee table.
- Existing rows will have NULL in the new column until updated.

Key Points / Advantages:

- Allows **flexible modification** of table structure.
- Avoids the need to **drop and recreate** tables for small changes.
- Can add **multiple columns** in a single ALTER TABLE statement if needed.

(b) Define indexing. How will it be useful for searching data? (4 mark)

Introduction / Definition:
Indexing is a database technique used to **speed up data retrieval**

from a table. An index is a **data structure** (like a pointer or a table of keys) that allows the DBMS to **locate records quickly** without scanning the entire table. It is similar to an index in a book, which helps find information without reading every page.

How Indexing Works:

- When a table is indexed on a column, the DBMS creates a **separate data structure** storing the column values along with **pointers to the corresponding rows** in the table.
- During a query, instead of scanning all rows, the DBMS searches the **index structure** to directly find the required data.
- Common types of indexes include **single-column index, composite index, unique index, and clustered/non-clustered index**.

Uses / Advantages for Searching Data:

- **Faster Search:** Reduces the time to locate rows, especially in **large tables**.
- **Efficient Query Execution:** Improves performance of SELECT, WHERE, and JOIN queries.
- **Supports Sorting:** Helps in **ORDER BY** and **GROUP BY** operations.
- **Reduces Disk I/O:** DBMS reads fewer blocks, saving memory and processing time.

Example:

Suppose we have a table Employee(eid, ename, salary) and we frequently search by eid.

CREATE INDEX idx_eid ON Employee(eid);

- Now, a query like SELECT * FROM Employee WHERE eid = 101; will **use the index** to quickly locate the row instead of scanning the entire table.

Key Points:

- Indexing improves **query performance** but may **slightly slow down insert, update, and delete operations** because the index must also be updated.
- Proper use of indexes is critical for **database optimization**.

(c) How can we measure cost of Join query? Explain using example. (7 mark)

Introduction / Definition:
In relational databases, **join operations** combine rows from two or more tables based on a related column. Join queries are fundamental but can be **resource-intensive**, especially for large tables. Measuring the **cost of a join query** is essential for **query optimization**. The cost generally refers to the **amount of I/O, CPU, and memory resources** needed to execute the join.

Factors Affecting Join Cost:

1. **Number of tuples (rows)** in each table.
2. **Size of each tuple** (number of attributes and data types).
3. **Access method** (table scan, index scan).
4. **Join algorithm** used (nested loop, sort-merge, hash join).

5. **Available memory and buffer pages** for temporary storage.

Join Algorithms and Cost Estimation:

1. Nested Loop Join (NLJ):

- **Method:** For each row in the outer table, scan the inner table to find matching rows.
- **Cost Formula:**

$$\begin{aligned} &Cost \\ &= (Number\ o && f\ r && ows\ i \\ &\times (Cost\ t && o\ a && ccess\ i \end{aligned}$$

- **Example:**
  -
    - Table A: 1000 rows, Table B: 500 rows
    - If scanning B fully for each row in A, cost = 1000 × 500 = 500,000 row accesses.

---

2. Sort-Merge Join (SMJ):

- **Method:** Sort both tables on the join key, then merge them to find matching tuples.
- **Cost Formula:**

$$\begin{aligned} &Cost \\ &= Cost\ o && f\ s && orting \\ &+ Cost\ o && f\ m && ergin \end{aligned}$$

- **Example:**
  - Table A: 1000 rows, Table B: 500 rows

- Sort A and B, then merge → fewer comparisons than nested loop, especially for large tables.

---

3. Hash Join:

- **Method:** Build a hash table on the smaller table using the join key, then probe it with rows from the larger table.
- **Cost Formula:**
- Cost=Cost to build hash table+Cost to probe hash table
- **Example:**
  - Table A: 1000 rows (build hash), Table B: 500 rows (probe)
  - Efficient for **equality joins** and large datasets.

Steps to Measure Join Cost:

1. Determine **join algorithm** based on table sizes and indexes.
2. Estimate **number of I/O operations** needed to read tables.
3. Add **CPU cost** for comparisons and hashing/sorting.
4. Compute **total cost** = I/O cost + CPU cost + memory usage.
5. Compare costs for different algorithms to choose **optimal join plan**.

Example in Words:

Suppose we have two tables:

- **Employee(eid, ename, deptid)** → 1000 rows
- **Department(deptid, dname)** → 50 rows

Query:

SELECT e.ename, d.dname

FROM Employee e

JOIN Department d

ON e.deptid = d.deptid;

Cost Estimation:

- Using **nested loop join** with Department as outer:
  - Cost = 50 × 1000 = 50,000 row accesses
- Using **hash join** with Department as build table:
  - Cost = build hash on 50 rows + probe 1000 rows → much lower than nested loop.

Key Points / Advantages of Measuring Join Cost:

- Helps **query optimizer** choose the most efficient execution plan.
- Reduces **query execution time** and resource usage.
- Essential for **large databases** and multi-user systems.

OR

Q.3 (a) Write an SQL command to modify the content of table. (3 mark)

Introduction / Definition:
In SQL, the **content of a table** can be modified using the UPDATE command. This command allows changing **existing data** in one or more rows of a table based on a specified condition. It is an essential part of **data manipulation operations (DML)** in a relational database.

Syntax:

UPDATE table_name

SET column1 = value1, column2 = value2, ...

WHERE condition;

Explanation of Syntax:

- UPDATE table_name → Specifies the table whose data needs to be modified.
- SET column1 = value1, ... → Lists the columns to be updated and the new values.
- WHERE condition → Optional; specifies which rows to update. If omitted, **all rows** will be updated.

Example:

Suppose we have a table Employee(eid, ename, salary, deptid) and we want to **increase the salary of employee with eid = 101**:

UPDATE Employee

SET salary = 75000

WHERE eid = 101;

Explanation:

- Updates the salary column for the employee whose eid is 101.
- Other rows in the table remain unaffected.

Key Points / Advantages:

- Allows **selective updating** of records.

- Supports **updating multiple columns** at once.
- Helps maintain **current and accurate data** in the database.

(b) Explain sparse indexing with the help of diagram. (4 mark)

Indexing in databases is a technique to speed up data retrieval. A **sparse index** is a type of **non-dense indexing** in which **index entries are not created for every record in the data file**, but only for **some records**, usually one per data block. Sparse indexing reduces the **size of the index** and **speeds up search** while using less storage.

Explanation:

- In a sparse index, the index stores **pointers to only some records** (typically the first record of each block).
- When searching, the DBMS uses the sparse index to **locate the block** and then scans that block to find the exact record.
- Sparse indexing is efficient for **large files** where creating a dense index for every record would be costly.
- Typically used with **sequentially ordered data**.

Advantages:

- Requires **less storage** than dense indexing.
- Faster **index creation and maintenance**.
- Reduces **disk I/O** compared to scanning the entire table.

Disadvantages:

- Searching may require **scanning within a block** after finding the block via the index.
- Slightly slower than dense indexing for exact record retrieval.

Example / Diagram (Described in Words):

Consider a data file with records sorted by Emp_Id:

Data File (Blocks):

- Block 1 → Emp_Id: 101, 102, 103
- Block 2 → Emp_Id: 104, 105, 106
- Block 3 → Emp_Id: 107, 108, 109

Sparse Index:

- Index entry → Points to the **first record in each block**:
  - 101 → Block 1
  - 104 → Block 2
  - 107 → Block 3

Search Process:

1. To find Emp_Id 105 → Check sparse index → 104 points to Block 2
2. Scan Block 2 → Locate 105

(Imagine a table of Index → Block pointer mapping on the left and data blocks on the right.)

(c) Draw and explain the query processing steps (7 mark)

Introduction / Definition:
Query processing in a Database Management System (DBMS) is the process of **translating a high-level SQL query into an efficient execution plan** that retrieves the desired data from the database. It ensures **correctness, efficiency, and optimization** of queries. Query processing involves several systematic steps to minimize **CPU, I/O, and memory costs**.

Steps of Query Processing:

1. Parsing and Translation:

- The SQL query is **parsed** to check **syntax and validity**.
- **Semantic checks** ensure tables, attributes, and data types exist.
- The query is translated into an **internal representation**, typically **relational algebra**.
- **Example:** SELECT ename FROM Employee WHERE deptid=10;
  → Translated to relational algebra:

$$\pi_{ename}(\sigma_{deptid=10}(Employee))$$

---

2. Query Optimization:

- The internal representation is passed to the **query optimizer**.
- The optimizer generates **equivalent query plans** using **algebraic transformations**.
- Selects the **most efficient execution plan** based on:
  - Available indexes
  - Join order
  - Estimated cost (I/O and CPU)

---

3. Plan Generation / Evaluation:

- The optimized query plan is converted into a **sequence of low-level operations** that the DBMS can execute.
- Operations include **table scans, index scans, joins, projections, and selections**.

- The DBMS may also determine **temporary storage** requirements and **access paths**.

---

4. Query Execution:

- The **execution engine** carries out the plan step by step.
- Data is retrieved, joined, filtered, and projected as per the plan.
- Final result is returned to the user or application.

Flow Diagram (Described in Words):

1. **User Query (SQL)** →
2. **Parsing & Translation** → Check syntax and convert to relational algebra →
3. **Query Optimization** → Generate multiple plans → Choose optimal plan →
4. **Plan Generation** → Convert plan to executable operations →
5. **Query Execution** → Retrieve data → Return result

(Imagine a flow from top to bottom with arrows connecting the above steps, showing SQL input to final output.)

Key Points / Advantages:

- Ensures **efficient query execution** and **minimal resource usage**.
- Helps in **cost estimation** and selecting the **best join/order strategy**.
- Supports **large databases** with multiple users and complex queries.

Example in Words:

- SQL Query: SELECT ename, salary FROM Employee WHERE salary > 50000;
- **Parsing:** Checks table Employee and columns exist.
- **Optimization:** Chooses to use an index on salary (if available) instead of full scan.
- **Execution:** Retrieves only rows with salary > 50000 and projects ename and salary.

Q.4 (a) What is relational model? Explain schema diagram using example. (3 mark)

Introduction / Definition:
The **relational model** is a way to organize data in a database using **tables (relations)**, where each table consists of **rows (tuples)** and **columns (attributes)**. It was proposed by **E.F. Codd** in 1970 and forms the foundation of most modern database systems. The relational model ensures **data integrity, flexibility, and ease of querying** using SQL.

Key Features:

- Data is stored in **relations (tables)**.
- Each row (tuple) represents a **unique record**.
- Each column (attribute) represents a **data field**.
- Relationships between tables are established using **primary and foreign keys**.
- Supports **operations like select, project, join** using relational algebra.

Relational Schema and Schema Diagram:

- A **relational schema** defines the **structure of a relation**: its name, attributes, and constraints.
- **Schema Diagram** is a **graphical representation** of relations and their relationships.

Example:

1. **Tables:**
    - **Employee(Emp_Id, Ename, Dept_Id, Salary)**
    - **Department(Dept_Id, Dname)**
2. **Relationships:**
    - Dept_Id in Employee is a **foreign key** referencing Department.

Schema Diagram (Described in Words):

- Rectangle → Employee with attributes: Emp_Id (PK), Ename, Dept_Id, Salary
- Rectangle → Department with attributes: Dept_Id (PK), Dname
- Line connecting Employee.Dept_Id → Department.Dept_Id indicating **foreign key relationship**
- **Interpretation:**
    - Each employee belongs to one department.
    - One department can have multiple employees.

Key Points / Advantages:

- Makes data **organized and structured**.

- Ensures **referential integrity** through primary and foreign keys.
- Easy to query using SQL.

(b) Define transaction and its properties? (4 mark)

Introduction / Definition:
A **transaction** in a Database Management System (DBMS) is a **logical unit of work** that consists of one or more database operations such as **INSERT, UPDATE, DELETE, or SELECT**. A transaction must be executed **completely or not at all** to maintain database consistency. Transactions are essential to **ensure reliability and integrity** in multi-user and concurrent environments.

Properties of a Transaction (ACID Properties):

1. Atomicity:

- Ensures that a transaction is **all-or-nothing**.
- If any part of the transaction fails, the **entire transaction is rolled back**.
- Example: Transferring money from one account to another – both debit and credit must succeed or fail together.

2. Consistency:

- Ensures that a transaction **transforms the database from one consistent state to another**.
- Integrity constraints and rules are preserved before and after the transaction.

3. Isolation:

- Ensures that **concurrent transactions** do not interfere with each other.

- Intermediate states of a transaction are **invisible to other transactions** until it commits.

4. Durability:

- Once a transaction commits, its changes are **permanent**, even in case of **system failure**.
- DBMS ensures committed data is safely stored on **non-volatile storage**.

Key Points / Advantages:

- Transactions **maintain data integrity and correctness**.
- Enable **safe concurrent access** in multi-user environments.
- Allow **recovery from failures** without loss of committed data.

(c) Write a note on view serializability (7 mark)

Introduction / Definition:
In a Database Management System (DBMS), **concurrent execution of transactions** is essential for **multi-user environments**. However, it can lead to **inconsistencies** if not managed properly. To ensure **correctness**, schedules of transactions are analyzed for **serializability**, which guarantees that the outcome of concurrent execution is the same as some **serial execution** of transactions.

View serializability is a **type of serializability** based on the **view of data items** accessed and modified by transactions. A schedule is **view serializable** if it is **equivalent to a serial schedule** in terms of **read and write operations on database items**.

Definition:

A schedule S is **view serializable** if there exists a serial schedule S' such that:

1. **Initial Reads are the Same:** If a transaction reads a data item initially, the corresponding transaction in the serial schedule also reads it initially.
2. **Read-From Relation is Preserved:** If a transaction reads a value written by another transaction in S, it reads the same value in S'.
3. **Final Writes are the Same:** The transaction that performs the final write of a data item in S also performs the final write in S'.

Steps / Conditions for View Serializability:

1. Identify **all read and write operations** of transactions.
2. Check which transaction **writes the initial value** of each item.
3. Check **which transaction reads values from others**.
4. Ensure that the **final writes** match with some serial execution.
5. If all conditions are met, the schedule is **view serializable**.

Example:

Transactions:

T1: R(A), W(A)

T2: R(A), W(A)

Schedule S:

R1(A), R2(A), W1(A), W2(A)

- **Read-From Relation:**

    - T1 reads initial value of A, writes it.
    - T2 reads initial value of A (before T1's write), writes it.
- **Final Write:**
    - W2(A) is the final write → matches some serial order T1 → T2.
- **Conclusion:**
    - Schedule S is **view serializable** as it can be transformed to a serial schedule **T1 → T2** preserving initial reads, read-from relations, and final writes.

Key Features:

- Focuses on **view of data** rather than strict order of operations.
- Allows **more concurrency** compared to conflict serializability.
- Useful for **optimistic concurrency control** techniques.

Advantages:

- Enables **safe concurrent execution** of transactions.
- Ensures **database consistency** without enforcing strict order of operations.

Disadvantages:

- Harder to **check automatically** compared to conflict serializability.
- Requires **analyzing read-from and final write relations** for all data items.

Applications:

- Multi-user database environments to **maximize concurrency**.
- Optimistic concurrency control methods in **distributed databases**.
- Ensuring **data consistency** without reducing transaction throughput.

OR

Q.4 (a) With the help of example explain Generalization.. (3 mark)

Generalization in DBMS

Introduction / Definition:
Generalization is a process in database design where **two or more lower-level entity sets** are combined into a **higher-level entity set** based on **common attributes**. It is the **reverse of specialization** and is used to **reduce redundancy** and create a more abstract view of data. Generalization helps in **simplifying the database structure** and representing **common characteristics** at a higher level.

Explanation:

- **Bottom-up approach:** Start from **specific entities** → combine them into a **general entity**.
- The general entity contains **common attributes** of all specialized entities.
- Specialized entities may have additional attributes unique to them.

Example:

- **Specific Entities:**
  - Car(Vehicle_Id, Model, Engine_Type)

    - Truck(Vehicle_Id, Model, Load_Capacity)
- **Generalized Entity:**
    - Vehicle(Vehicle_Id, Model) → captures **common attributes**
- **Interpretation:**
    - Car and Truck are **specializations** of Vehicle.
    - Vehicle represents the **generalized entity** with attributes shared by both.

Key Points / Advantages:

- Reduces **redundancy** by storing **common attributes once**.
- Simplifies **database design and maintenance**.
- Helps in **hierarchical organization** of entities.

(b) Define various approach used for Log based recovery (4 mark)

Introduction / Definition:
Log-based recovery is a method used in Database Management Systems (DBMS) to **restore the database to a consistent state** after a system crash or failure. The DBMS maintains a **log file** that records all **database modifications**. During recovery, the log is used to **redo committed transactions** and **undo uncommitted transactions**, ensuring **data integrity and durability**.

Approaches Used for Log-Based Recovery:

1. **UNDO Logging (Rollback Logging):**
    - Records **old values** of data items before they are modified.

- If a transaction fails or aborts, the DBMS **uses the log to restore old values**.
- Ensures **atomicity** by undoing incomplete transactions.
- **Example:** If T1 updates salary from 50,000 → 55,000 and fails, use log to revert to 50,000.

2. **REDO Logging (Forward Logging):**
   - Records **new values** of data items after they are modified.
   - If the system crashes **after commit but before changes are written to disk**, the DBMS **reapplies changes from the log**.
   - Ensures **durability** of committed transactions.
   - **Example:** T2 commits updating salary to 60,000; after crash, redo ensures value is 60,000.
3. **UNDO/REDO Logging (Combined Approach):**
   - Maintains both **old and new values** in the log.
   - Allows the DBMS to **undo uncommitted transactions** and **redo committed transactions**.
   - Provides **complete recovery** from system failures.
4. **Checkpoints:**
   - Periodically writes **all committed data and log info** to disk.
   - Reduces the number of log records to be scanned during recovery.
   - Speeds up **recovery process** after crash.

Key Points / Advantages:

- Ensures **Atomicity, Consistency, and Durability** of transactions.
- Enables **fast recovery** in case of system failures.
- Minimizes **data loss and inconsistencies**.

(c) List and explain various types of Locks used for concurrency control. (7 mark)

Introduction / Definition:
In a multi-user database environment, **concurrent transactions** may access and modify the same data simultaneously. This can lead to **inconsistencies and anomalies** such as lost updates, dirty reads, and uncommitted data. To manage this, DBMS uses **locks**, which are mechanisms that **restrict access to data items** during transaction execution. Locks ensure **concurrency control** and maintain **database consistency**.

Types of Locks:

1. Shared Lock (S-Lock / Read Lock):

- Allows a transaction to **read a data item** but not modify it.
- Multiple transactions can **simultaneously hold shared locks** on the same data item.
- Prevents **write operations** until all shared locks are released.
- **Example:** Transaction T1 reads a record; T2 can also read but cannot write until T1 releases the lock.

---

2. Exclusive Lock (X-Lock / Write Lock):

- Allows a transaction to **both read and write a data item**.

- Only **one transaction can hold an exclusive lock** on a data item at a time.
- Prevents **other transactions from reading or writing** the locked item until released.
- **Example:** T1 updates a record; T2 must wait until T1 commits or rolls back.

---

3. Intention Locks:

- Used in **multi-level locking** to indicate a transaction's intention to acquire **shared or exclusive locks** on lower-level items.
- Types include:
  - **Intention Shared (IS):** Intends to acquire shared locks on some subordinate items.
  - **Intention Exclusive (IX):** Intends to acquire exclusive locks on some subordinate items.
- Helps in **hierarchical locking** for tables, pages, and rows efficiently.

---

4. Read Lock / Write Lock (Row-Level Locking):

- **Read Lock:** Allows **reading only** at row level; other reads allowed, writes blocked.
- **Write Lock:** Allows **updating a specific row**; all other reads and writes are blocked until released.
- Ensures **fine-grained control** and improves **concurrency**.

---

5. Binary Lock:

- A **single lock bit** for a data item; either locked (1) or unlocked (0).
- Simple but **less flexible**, can reduce concurrency.

---

6. Shared-Exclusive Locks (Two-Phase Locking Protocol):

- Combines **shared and exclusive locks**.
- Transactions **acquire locks in growing phase** and **release in shrinking phase**.
- Ensures **conflict serializability** and prevents **lost updates**.

Key Points / Advantages:

- Locks prevent **inconsistent reads and writes** in concurrent transactions.
- Enable **conflict serializability** of schedules.
- Support **two-phase locking protocol** for database correctness.

Disadvantages:

- May lead to **deadlocks** if multiple transactions wait indefinitely.
- Requires **careful management** to avoid performance bottlenecks.

Q.5 (a) How can we print output in PL/SQL block? Give example. (3 mark)

Introduction / Definition:
In PL/SQL, a **block** is a unit of code that can include declarations, executable statements, and exception handling. To **display output** from a PL/SQL block, the **DBMS_OUTPUT.PUT_LINE** procedure

is commonly used. It allows printing messages or variable values to the console or output window.

Syntax:

DBMS_OUTPUT.PUT_LINE('message or value');

- 'message or value' → The text or variable value to be printed.
- Multiple calls to PUT_LINE print **each line separately**.

Example:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello, GTU Students!');
  DBMS_OUTPUT.PUT_LINE('Welcome to PL/SQL programming.');
END;
/
```

Explanation:

- The BEGIN...END block contains **executable statements**.
- Each DBMS_OUTPUT.PUT_LINE prints a line to the output.
- Output:

```
Hello, GTU Students!
Welcome to PL/SQL programming.
```

Key Points / Advantages:

- Useful for **debugging and testing** PL/SQL code.

- Can print **variables, expressions, or messages**.
- Multiple lines can be printed sequentially.

(b) Explain cursor and its types. (4 mark)

Introduction / Definition:
A **cursor** in PL/SQL is a **pointer that allows row-by-row processing of SQL query results**. It is used to **retrieve, manipulate, and process individual rows** from the result set of a SELECT statement. Cursors are essential when **multiple rows are returned**, and we need **control over each row**.

Types of Cursors:

1. Implicit Cursor:

- Automatically created by PL/SQL when a **SELECT INTO, INSERT, UPDATE, or DELETE** statement is executed.
- Suitable for **single-row queries**.
- PL/SQL automatically manages the cursor lifecycle (open, fetch, close).
- **Example:**

```
DECLARE
  v_salary NUMBER;
BEGIN
  SELECT salary INTO v_salary FROM Employee WHERE eid = 101;
  DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);
END;
/
```

- Here, the cursor is **implicit**, handled by PL/SQL internally.

---

2. Explicit Cursor:

- Defined by the programmer for **multi-row queries**.
- Requires **manual operations**: OPEN, FETCH, and CLOSE.
- Allows **row-by-row processing**.
- **Example:**

```
DECLARE
  CURSOR emp_cursor IS
    SELECT ename, salary FROM Employee;
  v_name Employee.ename%TYPE;
  v_salary Employee.salary%TYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_name, v_salary;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Employee: ' || v_name || ', Salary: '
|| v_salary);
  END LOOP;
  CLOSE emp_cursor;
END;
/
```

Key Points / Advantages:

- Cursors allow **processing one row at a time** in PL/SQL.
- Implicit cursors are **simple for single-row queries**.
- Explicit cursors are **flexible for multi-row queries**.
- Cursors provide **control over query execution** and enable **conditional processing**.

(c) Write SQL queries for the following (query 1,2 and 3 carry one mark/query 4 & 5 carry two marks) Employee(eid, ename, salary, deptid) Department(deptid, dname)

1. List name of all employee

2. List all the departments with id > 5

3. List sum of the salary all employees.

4. Department wise count the number of employees

5. Find the name of employee earning highest salary.

SQL Queries for Employee and Department Tables

Given Tables:

- Employee(eid, ename, salary, deptid)
- Department(deptid, dname)

---

1. List name of all employees (1 Mark)

```
SELECT ename
FROM Employee;
```

Explanation:

- Uses SELECT to fetch the ename column from the Employee table.

---

2. List all the departments with id > 5 (1 Mark)

```
SELECT *
FROM Department
WHERE deptid > 5;
```

Explanation:

- Fetches all columns from Department where deptid is greater than 5.

---

3. List sum of the salary of all employees (1 Mark)

```
SELECT SUM(salary) AS total_salary
FROM Employee;
```

Explanation:

- Uses the aggregate function SUM to calculate **total salary** of all employees.

---

4. Department-wise count of the number of employees (2 Marks)

```
SELECT deptid, COUNT(*) AS emp_count
FROM Employee
GROUP BY deptid;
```

Explanation:

- Groups employees by deptid and counts the number of employees in each department.

- GROUP BY is used for **aggregate calculations per department**.

---

5. Find the name of employee earning the highest salary (2 Marks)

SELECT ename

FROM Employee

WHERE salary = (SELECT MAX(salary) FROM Employee);

Explanation:

- Subquery (SELECT MAX(salary) FROM Employee) finds the **highest salary**.
- Outer query fetches the employee(s) with that salary.
- Ensures **correct result even if multiple employees share the highest salary**.

Key Points:

- SQL aggregate functions like SUM, COUNT, and MAX are used for **calculations**.
- GROUP BY helps in **department-wise aggregation**.
- Subqueries allow **filtering based on computed values**.

OR

Q.5 (a) Define encryption and decryption. (3 mark)

Introduction / Definition:
Encryption and **decryption** are fundamental concepts in **cryptography**, which is used to secure data from unauthorized access. These techniques transform data between **readable (plaintext)** and

unreadable (ciphertext) forms to ensure **confidentiality and security**.

Encryption:

- Encryption is the process of **converting plaintext (original data)** into **ciphertext (encoded data)** using an **encryption algorithm and key**.
- Purpose: To **protect sensitive information** during storage or transmission.
- **Example:**
  - Plaintext: "HELLO"
  - Ciphertext (using a simple shift key 3): "KHOOR"

Decryption:

- Decryption is the **reverse process of encryption**, converting **ciphertext back to plaintext** using a **decryption algorithm and key**.
- Ensures that only **authorized users** can access the original data.
- **Example:**
  - Ciphertext: "KHOOR"
  - Plaintext (after decryption): "HELLO"

Key Points / Advantages:

- Ensures **data confidentiality** during storage and transmission.
- Forms the basis for **secure communication, digital signatures, and authentication**.

- Works with **symmetric keys** (same key for encryption and decryption) or **asymmetric keys** (public and private keys).

(b) List various types of Trigers. (4 mark)

Introduction / Definition:
A **trigger** is a **PL/SQL block** that automatically executes in response to a **specific event** on a table or view. Triggers are used to **enforce business rules, maintain integrity, or audit changes** in the database. They execute **automatically** without explicit invocation.

Types of Triggers:

1. BEFORE Trigger:

- Executes **before the triggering event** occurs (INSERT, UPDATE, DELETE).
- Often used to **validate or modify data** before it is written to the table.
- **Example:** Check if salary is above a minimum before inserting a record.

2. AFTER Trigger:

- Executes **after the triggering event** has occurred.
- Commonly used for **auditing, logging, or updating related tables**.
- **Example:** Log changes in a separate audit table after an update.

3. INSTEAD OF Trigger:

- Used mainly with **views**.
- Executes **instead of the triggering action** to allow **modification of views** that cannot be updated directly.

- **Example:** Update underlying table columns through a view.

4. ROW-Level Trigger:

- Executes **once for each row affected** by the triggering event.
- Useful for **row-specific validation or processing**.
- **Example:** Automatically calculate total salary per employee row during insert.

5. STATEMENT-Level Trigger:

- Executes **once per SQL statement**, regardless of the number of rows affected.
- Efficient for operations that are **not row-specific**, like logging the number of rows affected.

Key Points / Advantages:

- Enforces **business rules automatically**.
- Maintains **data integrity and auditing**.
- Reduces the need for **manual intervention** in repetitive tasks.

(c) What is role? How can we authorize a user using grant and revoke command? Give example. (7 mark)

Introduction / Definition:
In a Database Management System (DBMS), a **role** is a **named collection of privileges** that can be assigned to users. Roles simplify **user management and access control** by grouping related privileges together. Instead of granting individual privileges to each user, a role can be **granted once and assigned to multiple users**, making administration more efficient.

Granting Privileges Using Roles

- **GRANT** command is used to **assign privileges or roles** to users or other roles.
- Privileges can include **SELECT, INSERT, UPDATE, DELETE** on tables, or **system privileges** like CREATE TABLE.
- **Syntax:**

GRANT privilege_or_role TO user_name [WITH ADMIN OPTION];

Example:

1. Create a role:

CREATE ROLE manager_role;

2. Grant privileges to role:

GRANT SELECT, INSERT, UPDATE ON Employee TO manager_role;

3. Assign role to a user:

GRANT manager_role TO user1;

- **Explanation:** User user1 can now **select, insert, and update** Employee table because of the role.

Revoking Privileges Using Roles

- **REVOKE** command is used to **remove privileges or roles** from users or other roles.
- **Syntax:**

REVOKE privilege_or_role FROM user_name;

Example:

REVOKE manager_role FROM user1;

- **Explanation:** Removes all privileges associated with manager_role from user1.
- Useful to **control access** when users change roles or responsibilities.

Key Points / Advantages:

- **Simplifies privilege management:** Assigning roles is easier than granting individual privileges.
- **Improves security:** Privileges can be **controlled centrally** using roles.
- **Flexible:** Roles can be **granted to multiple users** and **revoke privileges easily**.